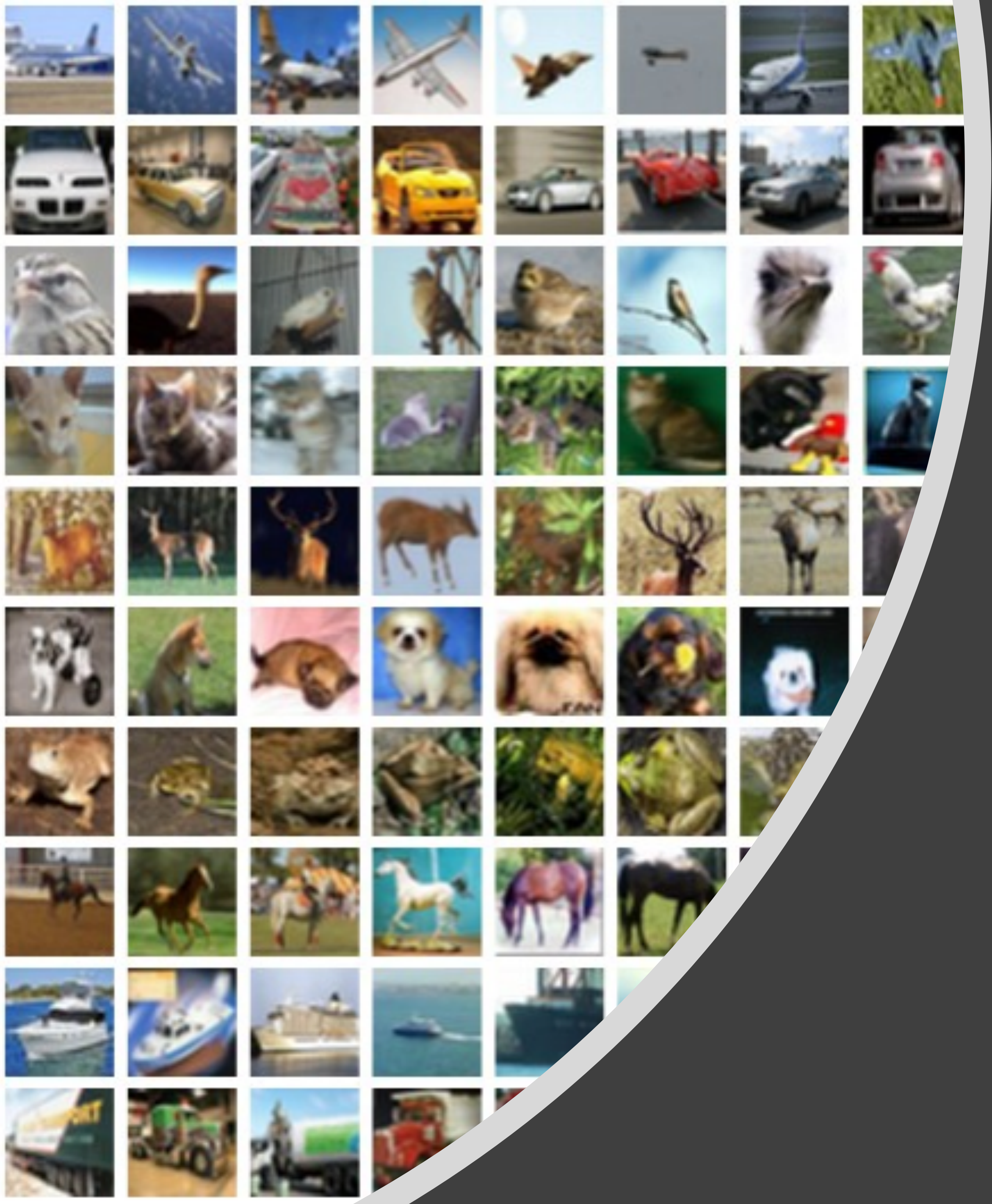
le



Dynamic Learning

Neural Networks and Backpropagation

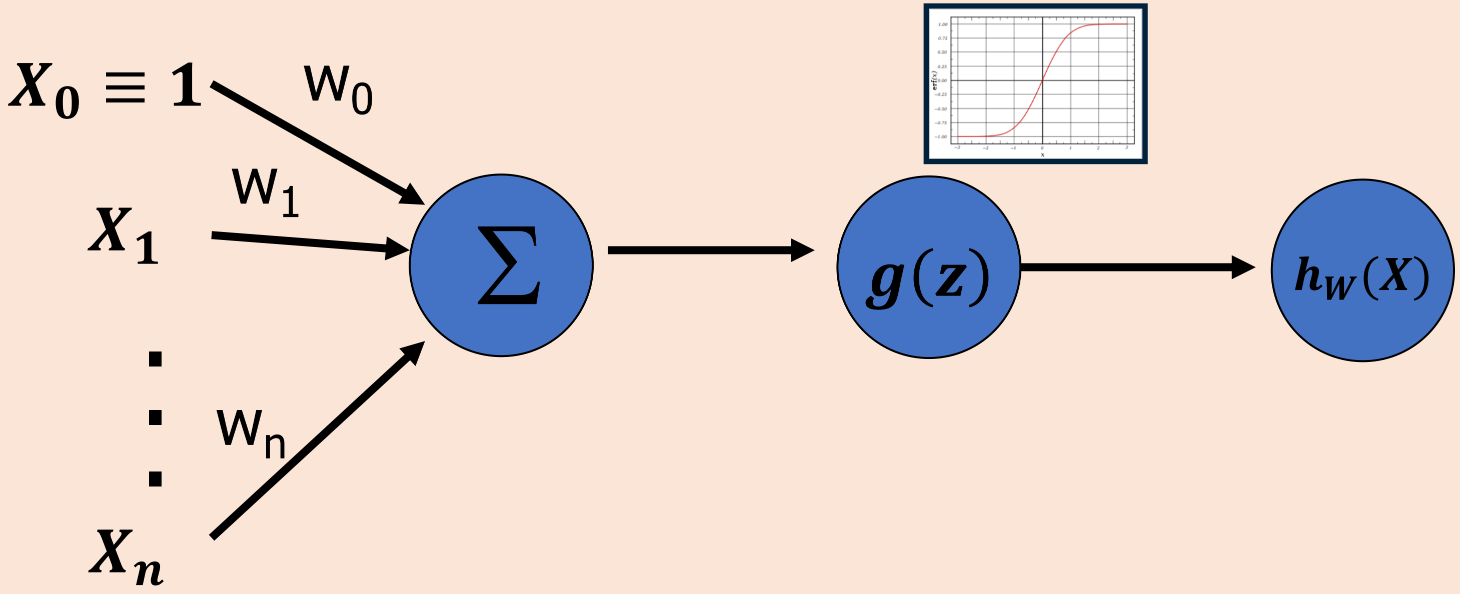
Plan for Class

- Perceptron Algorithm
  - Multi layered perceptron
- Regression
  - Linear
  - Logistic
- Gradient Descent
  - Batch Gradient Descent
  - Stochastic Gradient Descent
- Introduction to Neural Networks
  - Feedforward Neural Networks

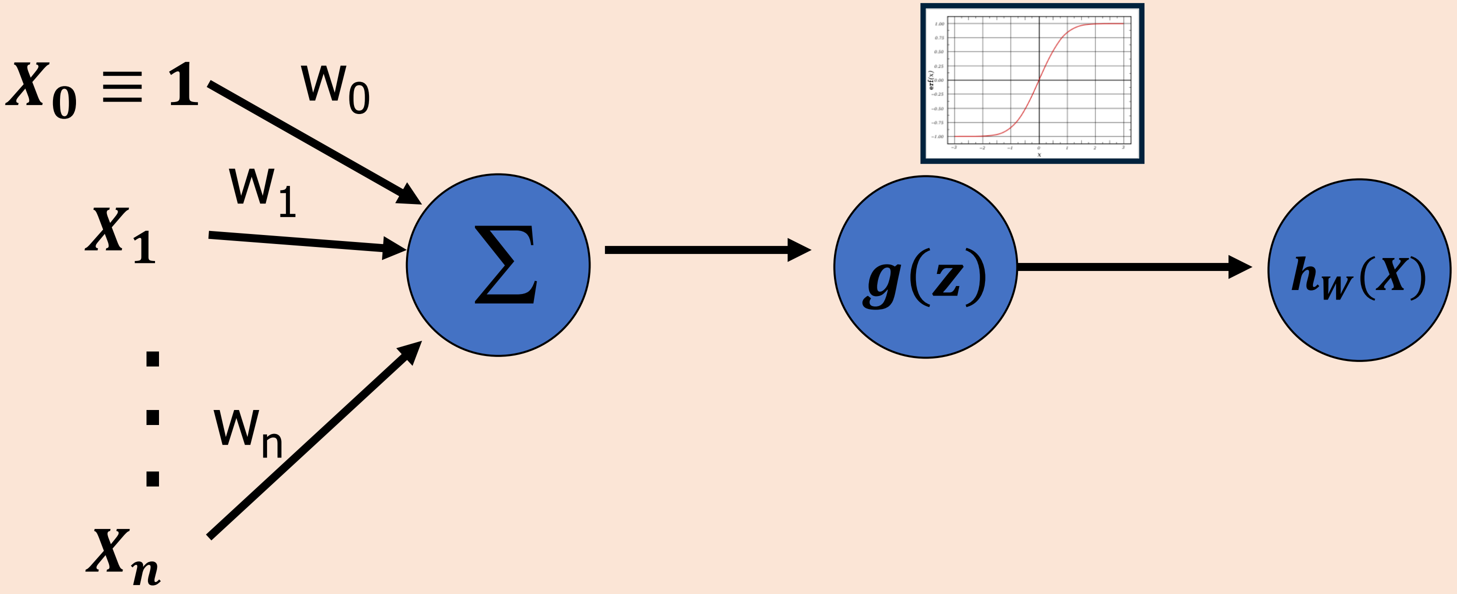
Logistic Regression

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$h_w(x) = g(\sum w_i x_i) = g(W \cdot X)$$



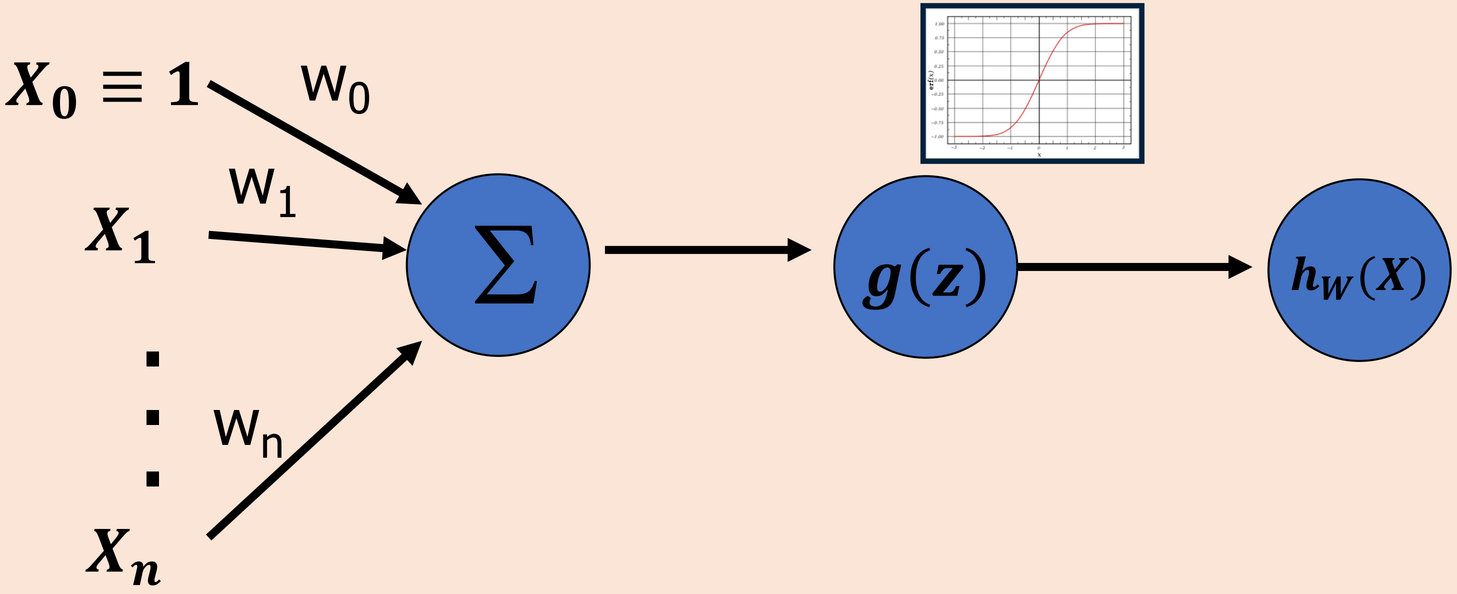
Input



Inputs To Neurons

- ❑ Arise from other neurons or from outside the network
- ❑ Nodes whose inputs arise outside the network are called *input nodes* and simply copy values
- ❑ An input may *excite* or *inhibit* the response of the neuron to which it is applied, depending upon the weight of the connection

Weights



Weights

- Represent synaptic efficacy and may be ***excitatory*** or ***inhibitory***
- Normally, positive weights are considered as ***excitatory*** while negative weights are thought of as ***inhibitory***
- ***Learning*** is the process of modifying the weights in order to produce a network that performs some function

Preparation

- **Training Set**
  A collection of input-output patterns that are used to train the network

- **Testing Set**
  A collection of input-output patterns that are used to assess network performance

- **Learning Rate-η**
  A scalar parameter, analogous to step size in numerical integration, used to set the rate of adjustments

Pseudo-Code Algorithm

- Randomly choose the initial weights
- While error is too large
  - For each training pattern (presented in random order)
    - **Propagate**
      - Apply the inputs to the network
      - Calculate the output for every neuron from the input layer, through the hidden layer(s), to the output layer
      - Calculate the error at the outputs
    - **Optimize**
      - Use the output error to compute error signals for pre-output layers
      - Use the error signals to compute weight adjustments
      - Apply the weight adjustments
  - Periodically evaluate the network performance

Gradient Descent

Given a training set $(X_0, y_0), \ldots (X_k, y_k)$

Assume:

$$P\left(y_j = 1 \middle| X_j, W\right) = h_W\left(X_j\right)^{y_j}(1 - h_W\left(X_j\right)^{1-y_j})$$

$$L(W) = \Pi_j P\left(y_j = 1 \middle| X_j, W\right) = \Pi_j h_W\left(X_j\right)^{y_j}(1 - h_W\left(X_j\right)^{1-y_j})$$

$$J(W) = \log L(W) = \sum_j y_j \log(h_W\left(X_j\right) + \left(1 - y_j\right)\log(1 - h_W(X_j)$$

$$\frac{\partial J}{\partial w_i}(X) = (y - h_W\left(X_j\right)x_i$$

using $\frac{\partial g}{\partial z} = g(z)(1 - g(z))$

Gradient Descent

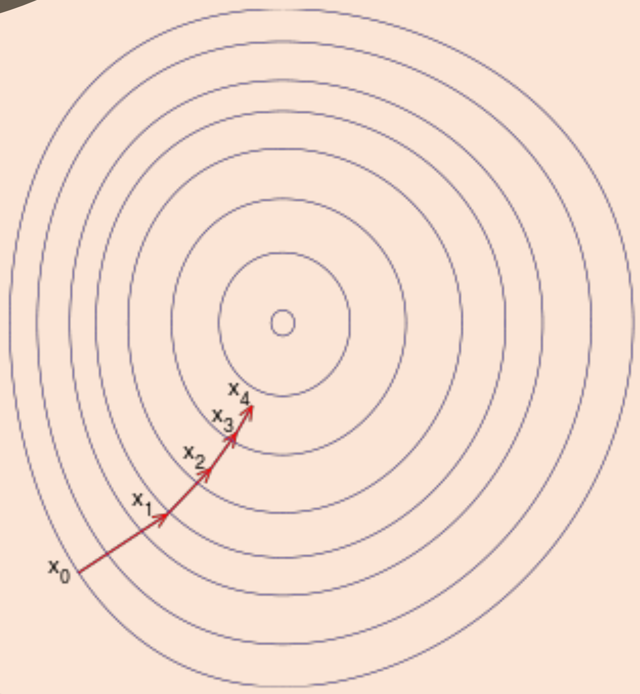
Given a training set $(X, y) \in (X_0, y_0), \ldots (X_k, y_k)$

$$W \leftarrow (w_{0,1}, \ldots w_{0,n})$$

$$for\ (X, y) \quad in \quad (X_0, y_0), \ldots (X_k, y_k):$$

$$h \leftarrow g(\sum w_i x_i)$$

$$w_i \leftarrow w_i + \eta \cdot \frac{\partial J}{\partial w_i}(X) = w_i + \eta \cdot x_i \cdot (y - h)$$



Propagate

```
In [13]: #Create a function that calculates the current SSE
         def propagate(w, b, X, Y):
             m = X.shape[1]
             A = sigmoid(np.dot(w.T,X)+b)
             cost = -1/m * np.sum(Y*np.log(A)+(1-Y)*np.log(1-A))
             dw = (1/m) * (np.dot(X,(A-Y).T))
             db = (1/m) * (np.sum(A-Y))
             cost = np.squeeze(cost)
             grads = {"dw": dw, "db": db}
             return grads, cost
```

Optimize

In [14]:

```
#Create a function that moves the estimates around and calculates the SSE to find optimal w and b
def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    costs = []
    for i in range(num_iterations):
        grads, cost = propagate(w,b,X,Y)
        dw = grads["dw"]
        db = grads["db"]
        w = w-learning_rate*dw
        b = b-learning_rate*db
        if i % 100 == 0:
            costs.append(cost)
        if print_cost and i % 100 == 0:
            print("Cost after iteration %i: %f" %(i, cost))

    params = {"w":w,"b":b}
    grads = {"dw":dw,"db":db}
    return params, grads, costs
```

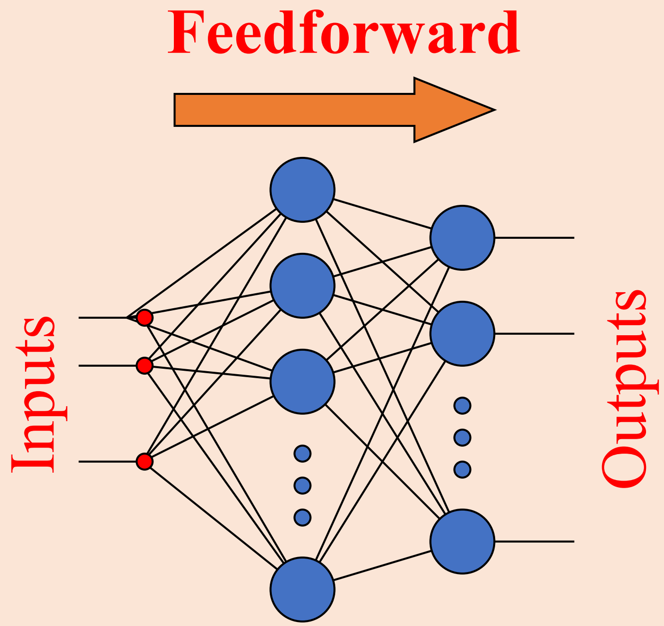
Predict

In [15]:
```
#------------------------Build the Logisitic Regression framework------------------------


def predict(w, b, X):
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0],1)
    A = sigmoid(np.dot(w.T,X)+b)
    for i in range(A.shape[1]):
        if A[0,i] <= 0.5:
            Y_prediction[0,i] = 0
        else:
            Y_prediction[0,i] = 1
    return Y_prediction
```

Feedforward Neural Networks

- Apply the value of each input parameter to each input node
- Input nodes compute only the identity function



Feedforward Neural Networks

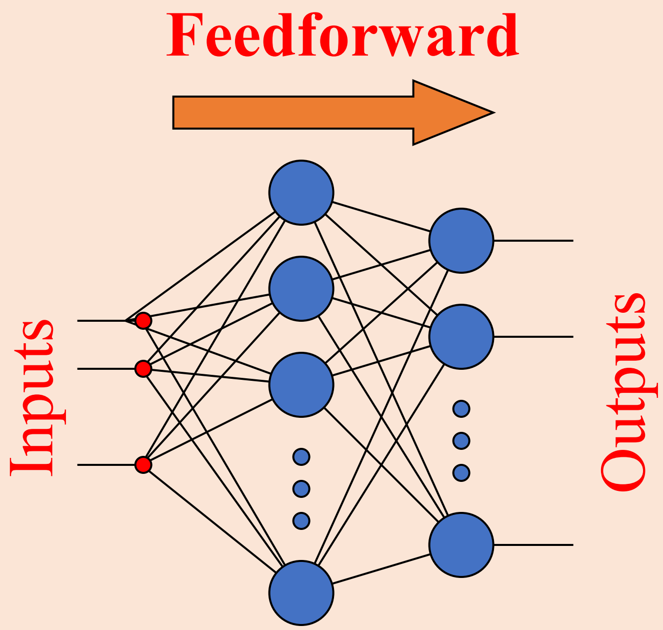
The output from neuron j for pattern p is O_{pj} where

$$O_{pj}(n_j) = g(-\lambda \cdot n_j)$$

and

$$n_j = bias \cdot W_{bias} + \sum_k O_{pk} \cdot W_{kj}$$

k ranges over the input indices and W_{jk} is the weight on the connection from input k to neuron j

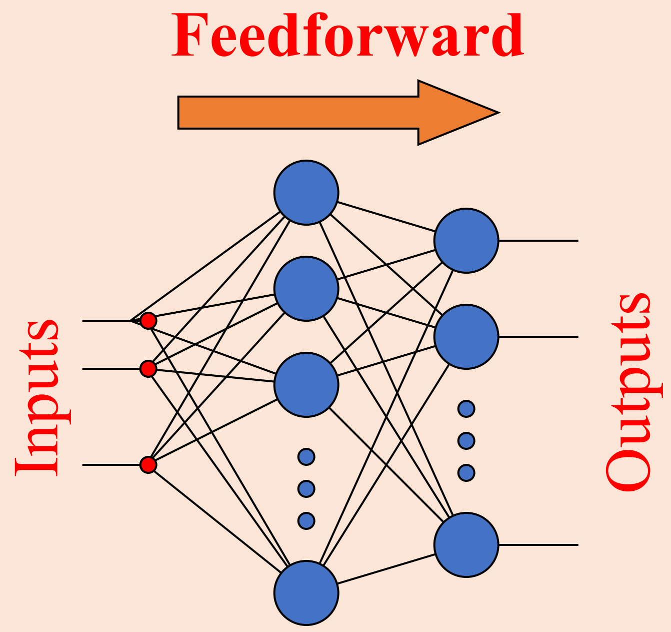


Error Signal For Each Output Neuron

- The output neuron error signal d_{pj} is given by

$$d_{pj}=(T_{pj}-O_{pj})\, O_{pj}\, (1-O_{pj})$$

- T_{pj} is the target value of output neuron j for pattern p
- O_{pj} is the actual output value of output neuron j for pattern p

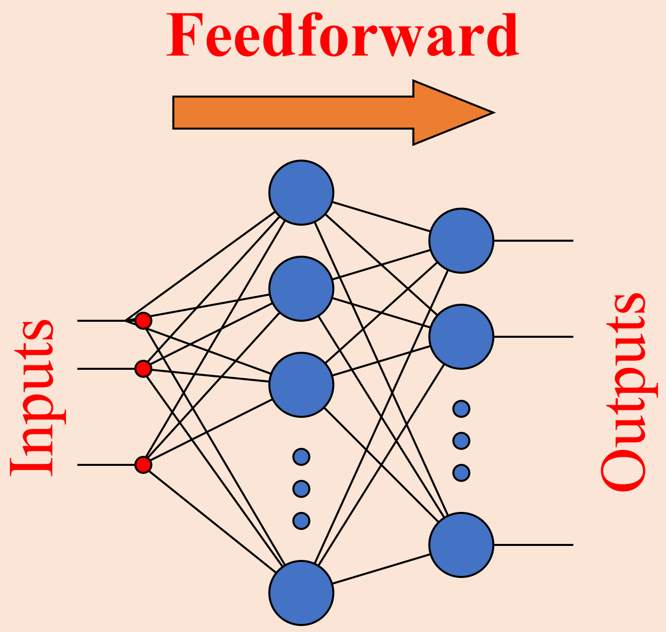


Error Signal For Each Output Neuron

- The hidden neuron error signal δ_{pj} is given by

$$\boldsymbol{\delta_{pj} = O_{pj}(1 - O_{pj}) \sum_{k} \delta_{pk} \cdot W_{kj}}$$

- where δ_{pk} is the error signal of a post-synaptic neuron k and W_{kj} is the weight of the connection from hidden neuron j to the post-synaptic neuron k



Error Signal For Each Output Neuron

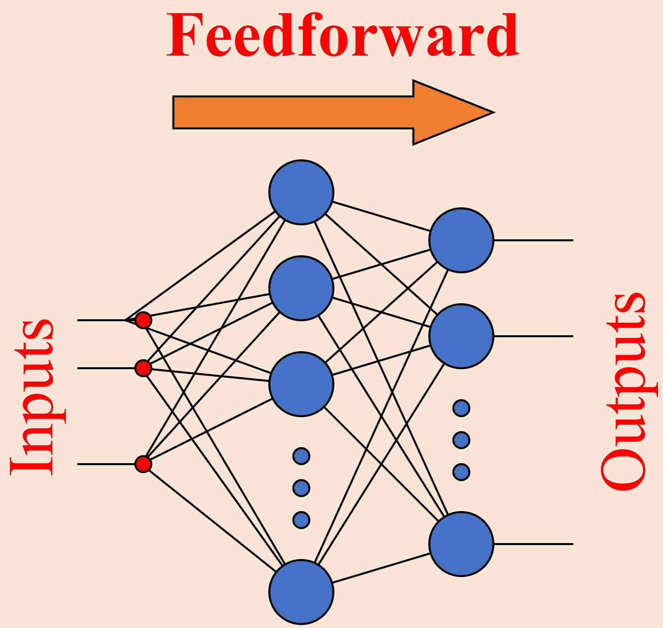
- Compute weight adjustments ΔW_{ji} at time t by

  $\Delta W_{ji}(t) = \eta\ \delta_{pj}\ O_{pi}$

- Apply weight adjustments according to

  $W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji}(t)$

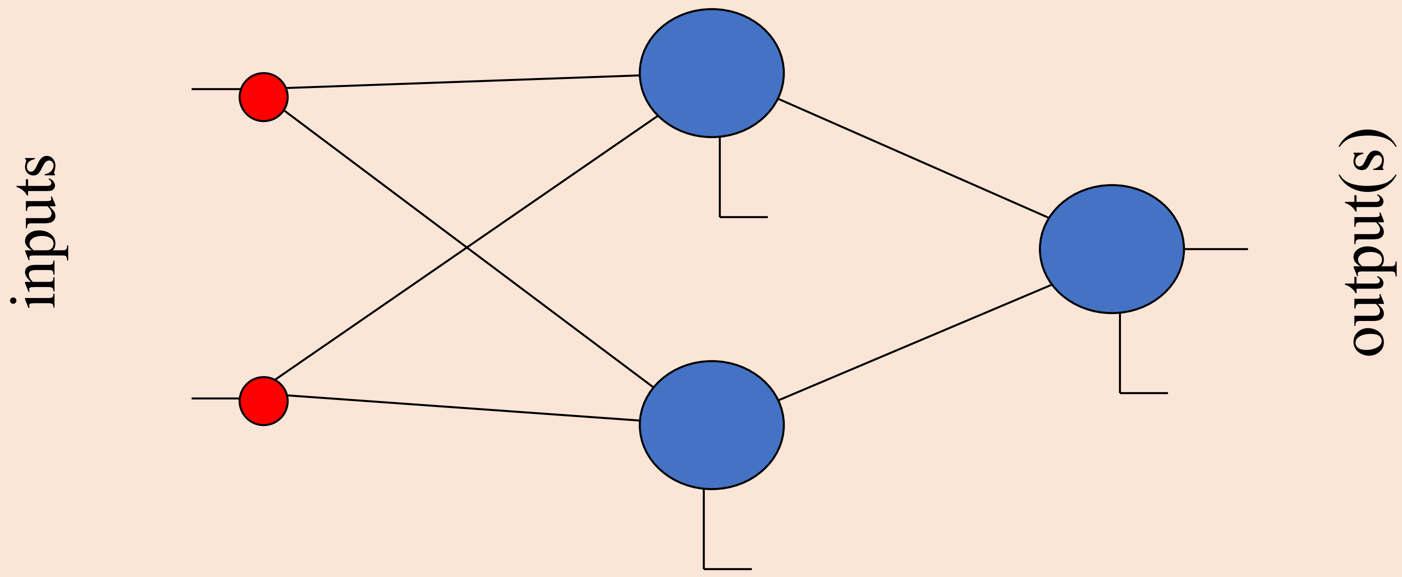
- *Some add a momentum term* $\alpha * \Delta W_{ji}(t-1)$



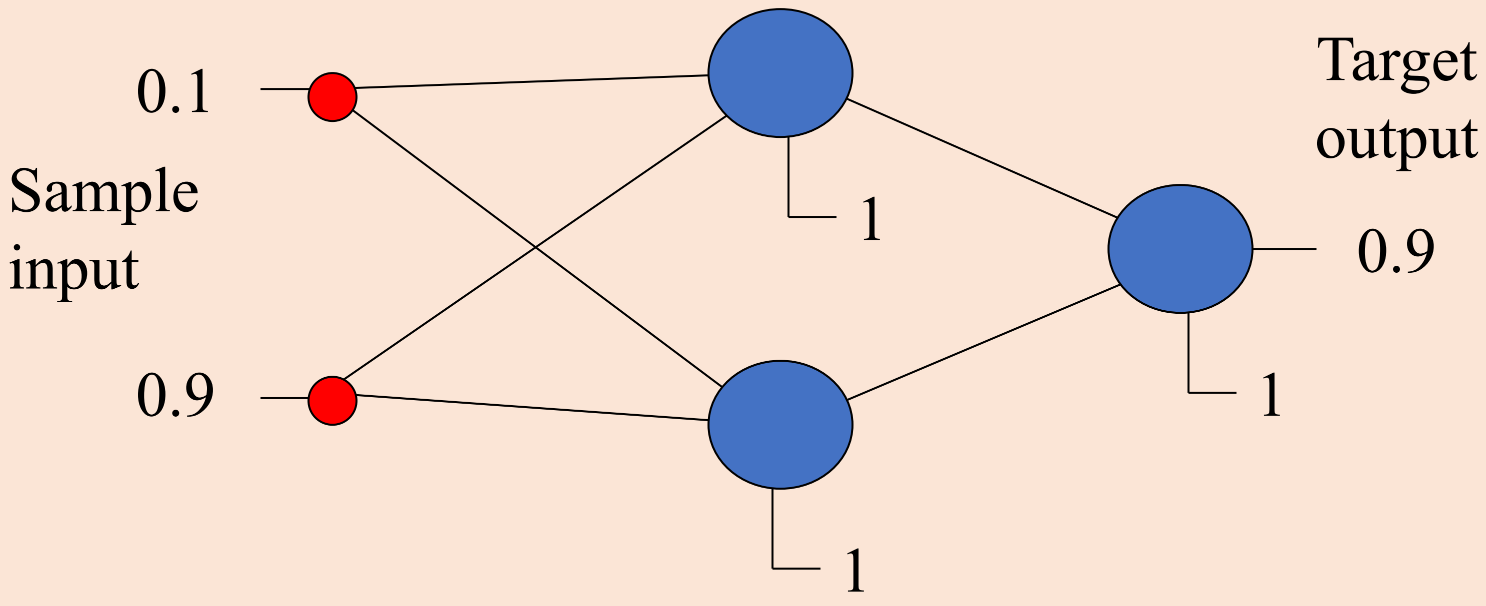
Example

- Training set
  - ((0.1, 0.1), 0.1)
  - ((0.1, 0.9), 0.9)
  - ((0.9, 0.1), 0.9)
  - ((0.9, 0.9), 0.1)
- Testing set
  - Use at least 121 pairs equally spaced on the unit square and plot the results
  - Omit the training set (if desired)

Example



Example



Feedforward Network Training by Backpropagation

- Select an architecture
- Randomly initialize weights
- While error is too large
  - Select training pattern and feedforward to find actual network output
  - Calculate errors and backpropagate error signals
  - Adjust weights
- Evaluate performance using the test set